Priority queue (heap)

Priority queue ADT

- Priority queue is an ADT that allows for at least two operations
- 1) insert
  - Equivalent of enqueue from ordinary queue
- 2) deleteMin
  - Equivalent of dequeue from ordinary queue

Example applications:

- OS scheduler (run highest priority process next)
- Network router
- Search (Dijkstra's, A* algorithm)

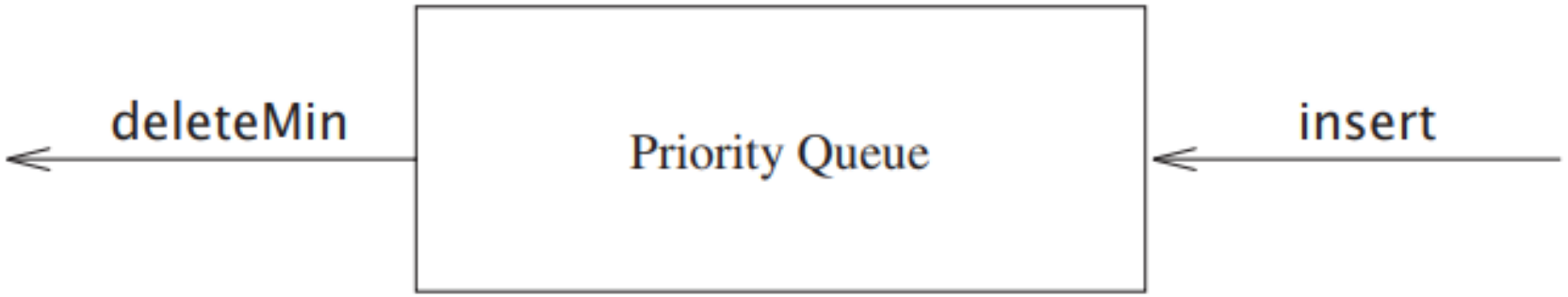

Figure 6.1 Basic model of a priority queue

Implementing a priority queue

- Many ways to implement priority queue ADT, complexity of insert and deleteMin will depend on what data structure we use
- Linked list
  - Insertion = $O(1)$ (simply add element to end of list)
  - deleteMin = $O(n)$ (traverse list to find and delete element)
- Array
  - Insertion = $O(1)$ (push_back)
  - deleteMin = $O(n)$ (iterate array to find min, delete, then move remaining elements)
- BST
  - Insertion = $O(logn)$ (assuming tree is balanced)
  - deleteMin = $O(logn)$ (assuming tree is balanced)

Binary heap (heap)

- The *heap* data structure allows us to implement a priority queue with `insert` in $O(1)$ and `deleteMin` in $O(logn)$
- **Structure property of heap:** a heap is a binary tree that is completely filled, with possible exception of bottom level, which is filled from left to right (also known as a *complete binary tree)*
- Complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes
- Because of this regular structure, can be represented as an array, and link structure is not necessary
- For any element in array position i, the left child is in position $2i$, right child is in cell after left child $(2i + 1)$ and parent is in $i/2$

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 6.3 Array implementation of complete binary tree

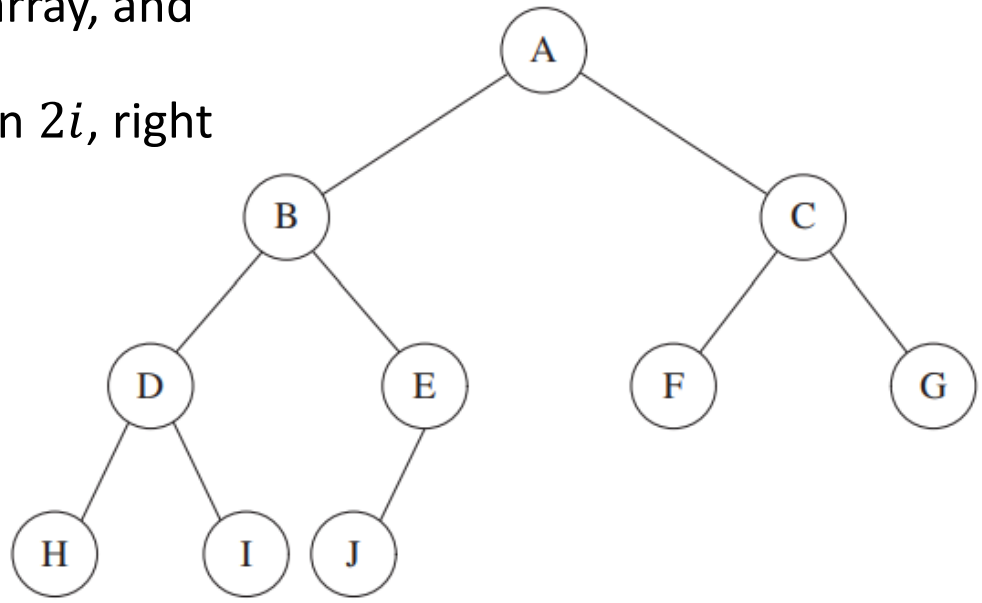

Figure 6.2 A complete binary tree

Heap-order property

- The *heap-order property* is what allows us to obtain $O(1)$ deleteMin and $O(logn)$ insert
- **Heap order property:** for every node X, the value in parent of X is smaller than or equal to the value in X, with the exception of the root which has no parent
- $\Rightarrow$ **Minimum element can always be found in constant time**

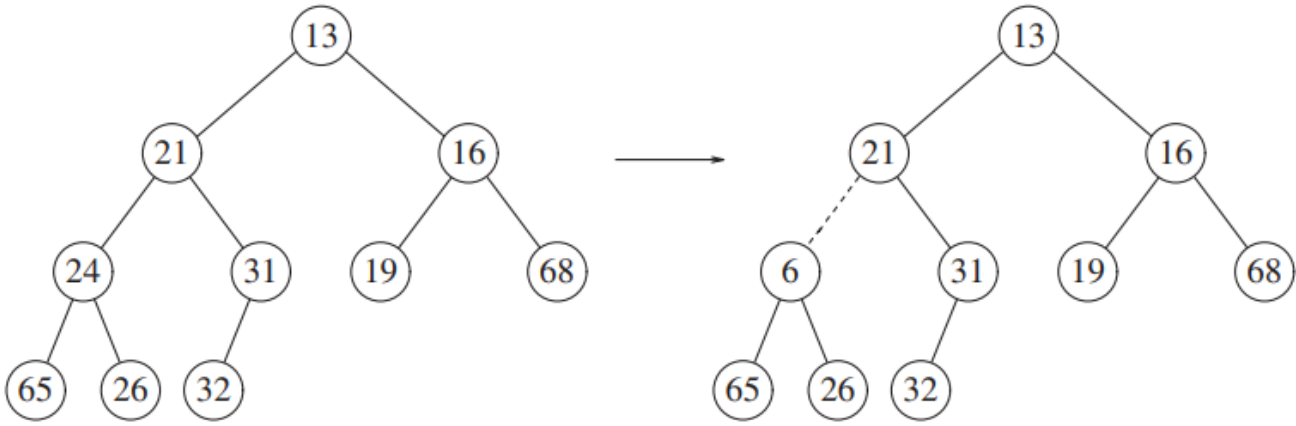

Figure 6.5 Two complete trees (only the left tree is a heap)

Basic heap operations: insert

- Insert: to insert element X into heap, create a hole in next available location (otherwise tree will not be complete)
- If X can be placed in hole without violating heap order property, we are done
- Otherwise, perform a "percolate up" operation by sliding down the element that is in the parent of the hole into the hole, thus "bubbling up" the hole towards the root
- The "percolate up" is continued until a location for the new element that does not violate heap order property is found
- Time complexity:$O(logn)$ worst-case, $O(1)$ in average-case. New element moves up by 1.607 levels in average case (based on empirical tests on random input)

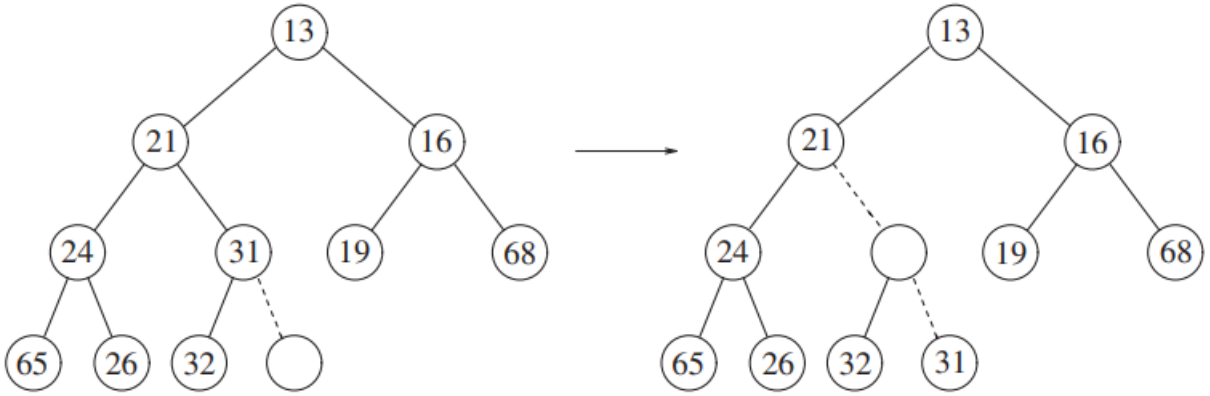

Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

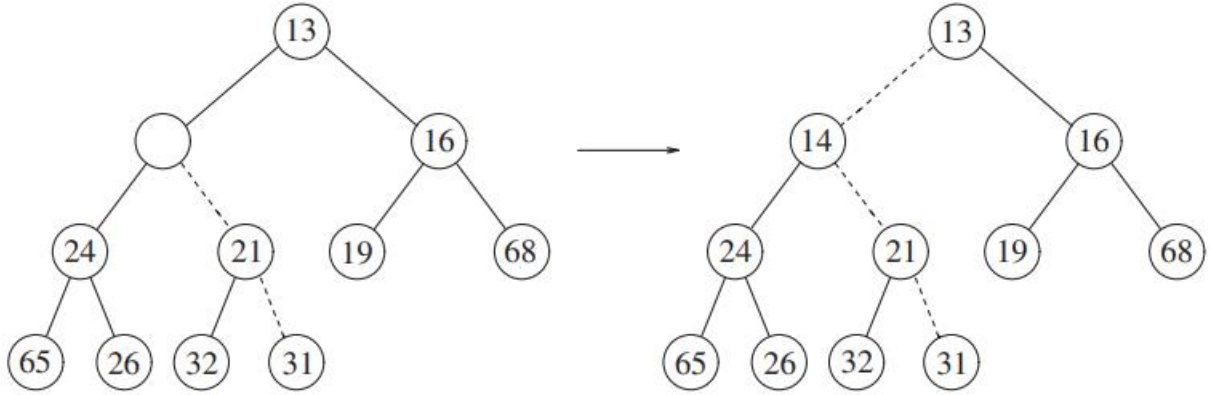

Figure 6.7 The remaining two steps to insert 14 in previous heap

Insert code

```
1       /**
2        * Insert item x, allowing duplicates.
3        */
4       void insert( const Comparable & x )
5       {
6           if( currentSize == array.size( ) - 1 )
7               array.resize( array.size( ) * 2 );
8
9               // Percolate up
10          int hole = ++currentSize;
11          Comparable copy = x;
12
13          array[ 0 ] = std::move( copy );
14          for( ; x < array[ hole / 2 ]; hole /= 2 )
15              array[ hole ] = std::move( array[ hole / 2 ] );
16          array[ hole ] = std::move( array[ 0 ] );
17      }
```

Figure 6.8 Procedure to insert into a binary heap

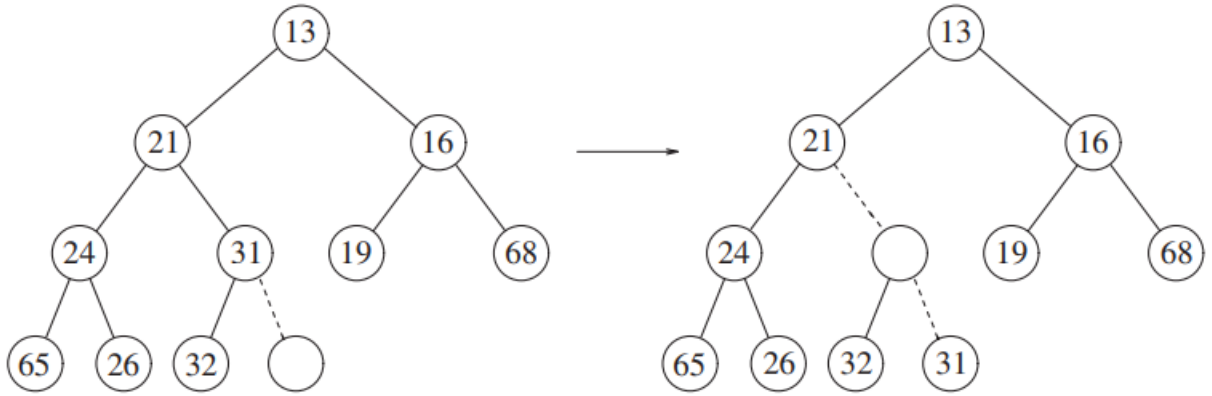

Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

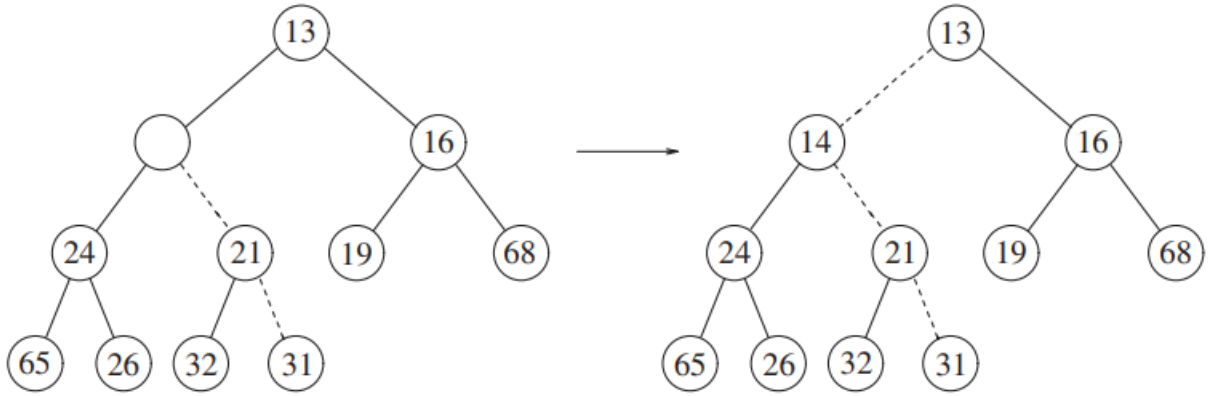

Figure 6.7 The remaining two steps to insert 14 in previous heap

deleteMin

- deleteMin handled similarly to insertion
- We already know where the min is (it's the root).
- Removing min creates a hole at the top of the tree
- To maintain completeness of binary tree, we take the last element from the bottom level and place it at top
- Then, we 'percolate down' this element until heap property is restored
- Average running time of $O(logn)$ because element placed at root is percolated down close to bottom of heap

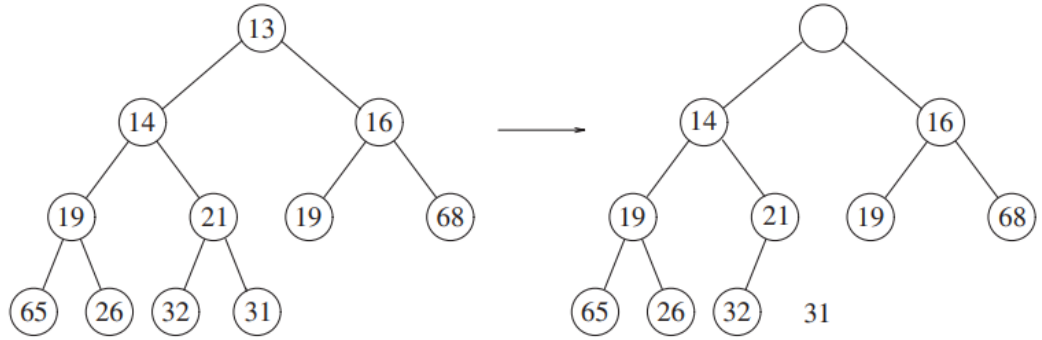

Figure 6.9 Creation of the hole at the root

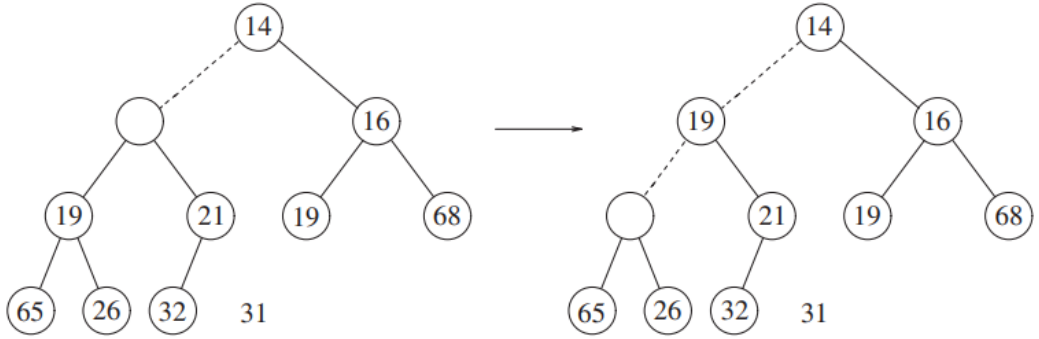

Figure 6.10 Next two steps in `deleteMin`

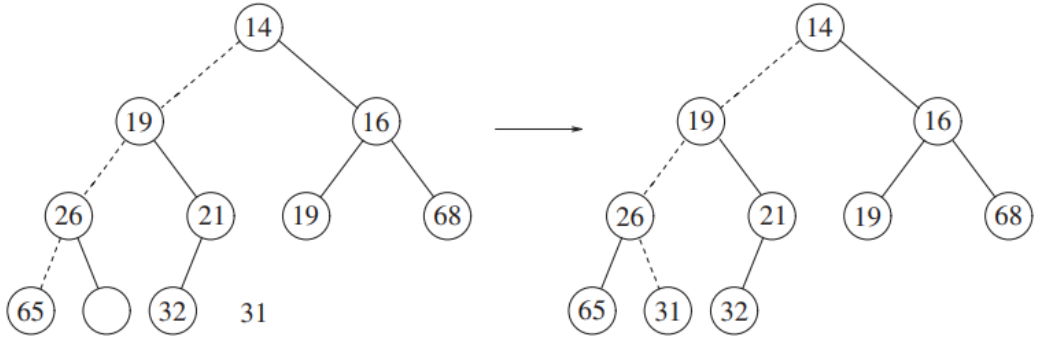

Figure 6.11 Last two steps in `deleteMin`

deleteMin

```
1       /**
2        * Remove the minimum item.
3        * Throws UnderflowException if empty.
4        */
5       void deleteMin( )
6       {
7           if( isEmpty( ) )
8               throw UnderflowException{ };
9
10          array[ 1 ] = std::move( array[ currentSize-- ] );
11          percolateDown( 1 );
12      }
13
14      /**
15       * Remove the minimum item and place it in minItem.
16       * Throws UnderflowException if empty.
17       */
18      void deleteMin( Comparable & minItem )
19      {
20          if( isEmpty( ) )
21              throw UnderflowException{ };
22
23          minItem = std::move( array[ 1 ] );
24          array[ 1 ] = std::move( array[ currentSize-- ] );
25          percolateDown( 1 );
26      }
```

```
28      /**
29       * Internal method to percolate down in the heap.
30       * hole is the index at which the percolate begins.
31       */
32      void percolateDown( int hole )
33      {
34          int child;
35          Comparable tmp = std::move( array[ hole ] );
36
37          for( ; hole * 2 <= currentSize; hole = child )
38          {
39              child = hole * 2;
40              if( child != currentSize && array[ child + 1 ] < array[ child ] )
41                  ++child;
42              if( array[ child ] < tmp )
43                  array[ hole ] = std::move( array[ child ] );
44              else
45                  break;
46          }
47          array[ hole ] = std::move( tmp );
48      }
```

Figure 6.12 Method to perform `deleteMin` in a binary heap

buildHeap

- Heap is sometimes constructed from an initial collection of items (instead of getting items over time, in a stream)
- Simple approach: insert items into heap one-by-one
  - $O(n)$ average case, $O(nlogn)$ worst-case
- Better approach:
  - 1) place all items into heap in any order, maintaining structure property but not heap-order property
  - 2) call percolateDown on every item in the unordered heap, starting at the root

```
1       explicit BinaryHeap( const vector<Comparable> & items )
2         : array( items.size( ) + 10 ), currentSize{ items.size( ) }
3       {
4           for( int i = 0; i < items.size( ); ++i )
5               array[ i + 1 ] = items[ i ];
6           buildHeap( );
7       }
8
9       /**
10       * Establish heap order property from an arbitrary
11       * arrangement of items. Runs in linear time.
12       */
13      void buildHeap( )
14      {
15          for( int i = currentSize / 2; i > 0; --i )
16              percolateDown( i );
17      }
```

Figure 6.14 buildHeap and constructor

buildHeap

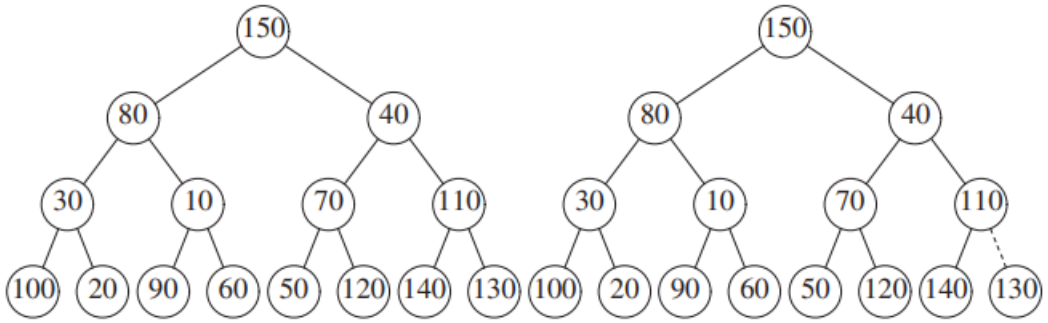

Figure 6.15 Left: initial heap; right: after `percolateDown(7)`

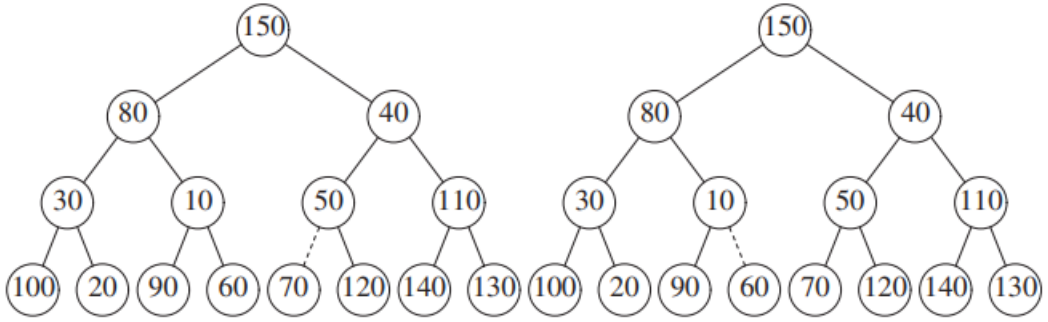

Figure 6.16 Left: after `percolateDown(6)`; right: after `percolateDown(5)`

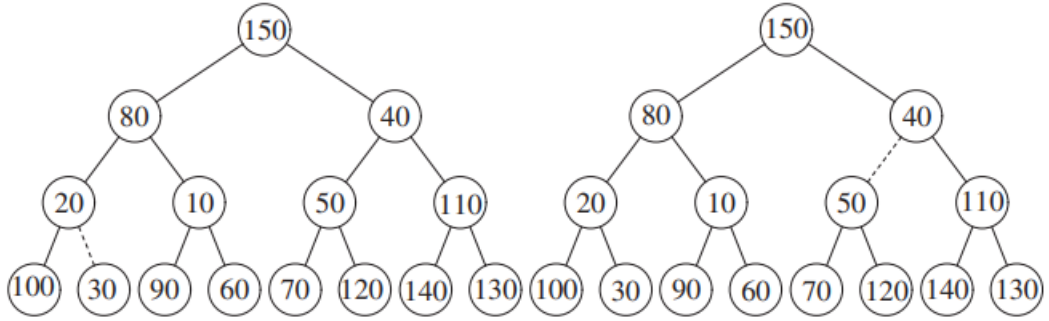

Figure 6.17 Left: after `percolateDown(4)`; right: after `percolateDown(3)`

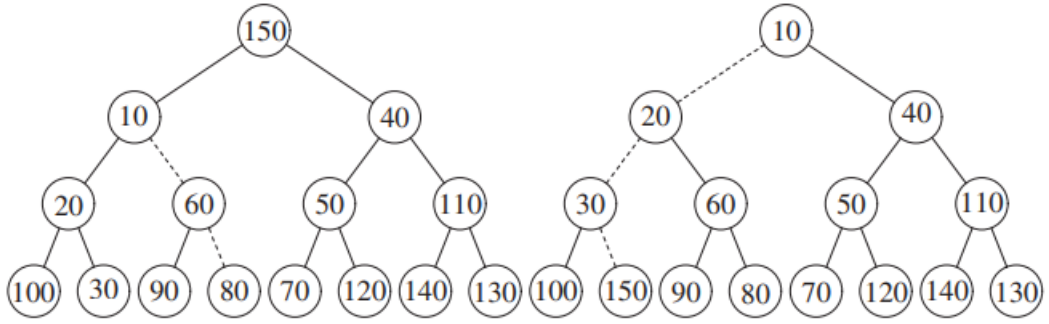

Figure 6.18 Left: after `percolateDown(2)`; right: after `percolateDown(1)`

6.15: unordered heap. The other 7 trees in 6.15 through 6.18 show the result of seven percolateDown operations. Each dashed line corresponds to two comparisons, one to find the smaller child, and one to compare the smaller child with the node. Notice there are only 10 dashed lines, corresponding to 20 comparisons

Time complexity of buildHeap

- To bound running time of buildHeap, must bound number of dashed lines. This can be done by computing sum of heights of all nodes in the heap, which is the maximum number of dashed lines. Need to show this is $O(N)$
- **Theorem:** for perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of heights of nodes is $2^{h+1} - 1 - (h + 1)$
- **Proof:** easy to see that tree consists of 1 node at height h, 2 nodes at height $h - 1$, 2^2 nodes at height $h - 2$ and in general 2^i nodes at height $h - i$.
- sum of heights of all nodes is then

$$S = \sum_{i=0}^{h} 2^i(h - i) = h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \cdots + 2^{h-1}(1)$$

- Multiplying by 2 gives the equation

$$2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \cdots + 2^h(1)$$

- Subtracting these two equations yields

$$S = -h + 2 + 4 + 8 + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h + 1)$$

- Which proves the theorem

Heapsort

- Build heap: $O(n)$
- Call removeMin n times (removeMin is $O(logn)$
- Total: $O(n) + O(nlogn) = O(nlogn)$

- When you do assignment 4, make sure to include time taken to build the heap/BST!

D-heaps

- Many other types of heap exist
- D-heap is identical to a binary heap, except all nodes have *d* children
- Advantage: shallower than binary heap, improving the running time of insert to $O(\log_d n)$
- deleteMin is slower, however because we need to do $d - 1$ comparisons to find the minimum of d children
- Also, using d to index into the array is slower than 2 because we can't use bit-shifting operations to get the indices
- Could be useful in applications where number of insertions is far greater than number of deleteMins

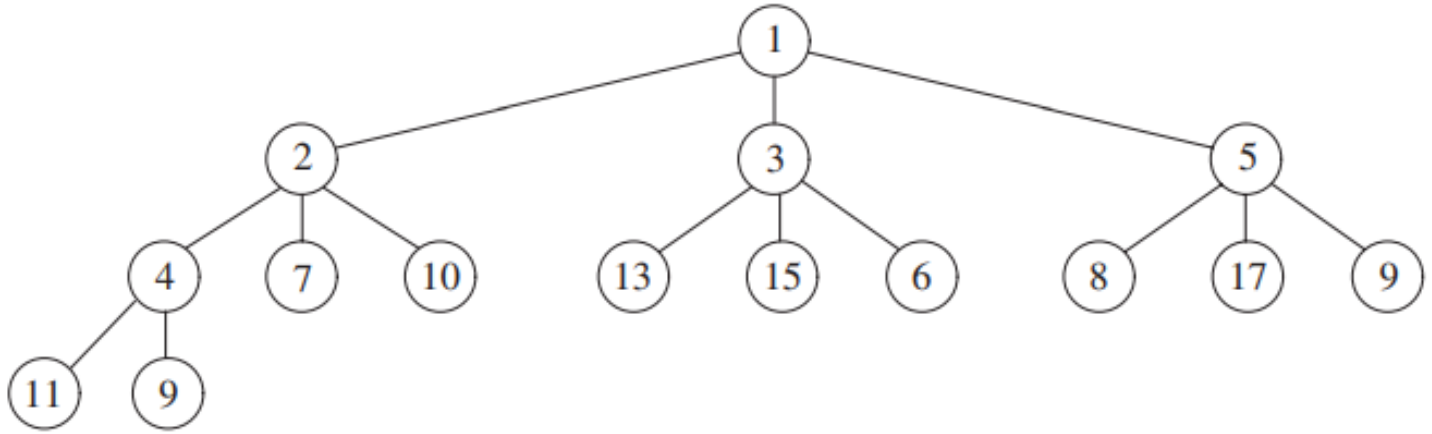

Figure 6.19 A d-heap ($d = 3$)

Other heap types

- Leftist heap
  - Able to merge quickly with other heaps
- Fibonacci heap
  - Better asymptotic running time for some operations than binary heap
- Skew heap
  - More balanced version of leftist heap
- Binomial queue
  - Another type of mergeable heap
- Etc.